Reuse Distance-Based Cache Hint Selection

Kristof Beyls and Erik H. D'Hollander

Department of Electronics and Information Systems
Ghent University
Sint-Pietersnieuwstraat 41,
9000 Ghent, Belgium
{kristof.beyls,erik.dhollander}@elis.rug.ac.be

Abstract. Modern instruction sets extend their load/store-instructions with cache hints, as an additional means to bridge the processor-memory speed gap. Cache hints are used to specify the cache level at which the data is likely to be found, as well as the cache level where the data is stored after accessing it. In order to improve a program's cache behavior, the cache hint is selected based on the data locality of the instruction. We represent the data locality of an instruction by its reuse distance distribution. The reuse distance is the amount of data addressed between two accesses to the same memory location. The distribution allows to efficiently estimate the cache level where the data will be found, and to determine the level where the data should be stored to improve the hit rate. The Open64 EPIC-compiler was extended with cache hint selection. Execution on an Itanium multiprocessor shows speedups of up to 36% in numerical and 23% in non-numerical programs.

1 Introduction

The growing speed gap between the memory and the processor push computer architects, compiler writers and algorithm designers to conceive ever more powerful data locality optimizations. However, many programs still stall more than half of their execution time, waiting for data to arrive from a slower level in the memory hierarchy. Therefore, the efforts of reducing memory stall time should be combined on the three different program levels: hardware, compiler and algorithm. In this paper, a combined approach at the compiler and hardware level is described.

Cache hints are emerging in new instruction set architectures. Typically they are specified as attachments to regular memory instructions, and occur in two kinds: source and target hints. The first kind, the *source cache specifier*, indicates at which cache level the accessed data is likely to be found. The second kind, the *target cache specifier*, indicates at which cache level the data is kept after the instruction is executed. An example is given in fig. 1, where the effect of the load instruction `LD_C2_C3` is shown. The source cache specifier `C2` suggests that at the start of the instruction, the data is expected in the L2 cache. The target cache specifier `C3` causes the data to be kept in the L3 cache, instead of keeping

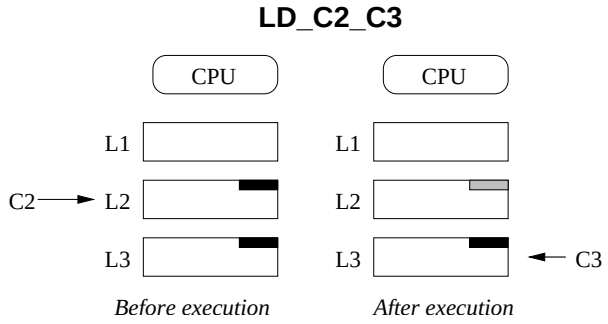


Fig. 1. Example of the effect of the cache hints in the load instruction `LD_C2_C3`. The source cache specifier `C2` in the instruction suggests that the data resides in the L2-cache. The target cache specifier `C3` indicates that the data should be stored no closer than the L3-cache. As a consequence, the data is the first candidate for replacement in the L2-cache.

it also in the L1 and L2 caches. After the execution, the data becomes the next candidate for replacement in the L2 cache.

In an Explicitly Parallel Instruction Computing architecture (EPIC), the source and destination cache specifiers are used in different ways.

The *source cache specifiers* are used by the compiler to know the estimated data access latency. Without these specifiers, the compiler assumes that all memory instructions hit in the L1 cache. Using the source cache specifier, the compiler is able determine the true memory latency of instructions. It uses this information to schedule the instructions explicitly in parallel.

The *target cache specifiers* are used by the processor, where they indicate the highest cache level at which the data should be kept. A carefully selected target specifier will maintain the data at a fast cache level, while minimizing the probability that it is replaced by intermediate accesses.

Small and fast caches are efficient when there is a high data locality, while for larger and slower caches lower data locality suffices. To determine the data locality, the reuse distance is measured. After measuring it, the reuse distance metric is used as a discriminating function to determine the most appropriate cache level and associated cache hints.

The reuse distance-based cache hint selection was implemented in an EPIC-compiler and tested on an Itanium multiprocessor. On a benchmark of general purpose and numerical programs, up to 36% speedup is measured, with an average speedup of 7%.

The emerging cache hints in EPIC instruction sets are discussed in sect. 2. The definition of the reuse distance, and some interesting lemmas are stated in sect. 3. The accurate selection of cache hints in an optimizing compiler is discussed in sect. 4. The experiments and results can be found in sect. 5. The related work is discussed in sect. 6. In sect. 7, the conclusion follows.

2 Software Cache Control in EPIC

Cache hints and cache control instructions are emerging in both EPIC[4, 7] and superscalar[6, 10] instruction sets. The most expressive and orthogonal cache hints can be found in the HPL-PD architecture[7]. Therefore, we use them in this work. The HPL-PD architecture defines 2 kinds of cache hints: *source cache specifiers* and *target cache specifiers*. An example of a load instruction can be found in fig. 1.

source cache specifier The source cache specifier indicates the highest cache level where the data is assumed to be found,

target cache specifier The target cache specifier indicates the highest cache level where the data should be stored. If the data is already present at higher cache levels, it becomes the primary candidate for replacement at those levels.

In an EPIC-architecture, the compiler is responsible for instruction scheduling. Therefore, the source cache specifier is used inside the compiler to obtain good estimates of the memory access latencies. Traditional compilers assume L1 cache hit latency for all load instructions. The source cache specifier allows the scheduler to have a better view on the latency of memory instructions. In this way, the scheduler can bridge the cache miss latency with parallel instructions. After scheduling, the source cache specifier is not needed anymore.

The target cache specifier is communicated to the processor, so that it can influence the replacement policy of the cache hierarchy. Since the source cache specifier is not used by the processor, only the target cache specifier needs to be encoded in the instruction. As such, the IA-64 instruction set only defines target cache specifiers. Our experiments are executed on an IA-64 Itanium-processor, since it is the only processor available with this rich set of target cache hints. E.g., in the IA-64 instruction set, the target cache hints `_C1`, `_C2`, `_C3`, `_C4` are indicated by the suffixes `.t1`, `.nt1`, `.nt2`, `.nta`[4]. Further details about the implementation of those cache hints in the Itanium processor can be found in [12].

In order to select the most appropriate cache hints, the locality of references to the same data is measured by the reuse distance.

3 Reuse Distance

The reuse distance is defined within the framework of the following definitions. When data is moved between different levels of the cache hierarchy, a complete cache line is moved. To take this effect into account when measuring the reuse distance, a memory line is considered as the basic unit of data.

Definition 1. *A* memory line[2] *is an aligned cache-line-sized block in the memory. When data is loaded from the memory, a complete memory line is brought into the cache.*

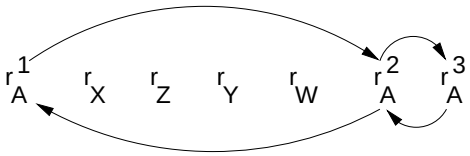


Fig. 2. A short reference stream with indication of the reuses. The subscript of the references indicates which memory line the reference accesses. The references r_X, r_Z, r_Y and r_W are not part of a reuse pair, since memory lines W, X, Y and Z are accessed only once in the stream. Reuse pair $\langle r_A^1, r_A^2 \rangle$ has reuse distance 4, while the reuse pair $\langle r_A^2, r_A^3 \rangle$ has reuse distance 0. The forward reuse distance of r_A^1 is 4, its backward reuse distance is ∞. The forward reuse distance of r_A^2 is 0, its backward reuse distance is 4.

Definition 2. *A* reuse pair $\langle r_1, r_2 \rangle$ *is a pair of references in the memory reference stream, accessing the same memory line, without intermediate references to the same memory line. The set of reuse pairs of a reference stream s is denoted by $\mathcal{R}_s$.*

Definition 3. *The* reuse distance *of a reuse pair $\langle r_1, r_2 \rangle$ is the number of unique memory lines accessed between references r_1 and r_2.*

Corollary 1. *Every reference in a reference stream s occurs at most 2 times in $\mathcal{R}_s$: once as the first element of a reuse pair, once as the second element of a reuse pair.*

Definition 4. *The* forward reuse distance *of a memory access x is the reuse distance of the pair $\langle x, y \rangle$. If there is no reuse pair where x is the first element, its forward reuse distance is ∞. The* backward reuse distance *of x is the reuse distance of $\langle w, x \rangle$. If there is no such pair, the backward reuse distance is ∞.*

Example 1. Figure 2 shows two reuse pairs in a short reference stream.

Lemma 1. *In a fully associative LRU cache with n lines, a reference with backward reuse distance $d < n$ will hit. A reference with backward reuse distance $d \geq n$ will miss.*

Proof. In a fully-associative LRU cache with n cache lines, the n most recently referenced memory lines are retained. When a reference has a backward reuse distance d, exactly d different memory lines were referenced previously. If $d \geq n$, the referenced memory line is not one of the n most recently referenced lines, and consequently will not be found in the cache. □

Lemma 2. *In a fully associative LRU cache with n lines, the memory line accessed by a reference with forward reuse distance $d < n$ will stay in the cache until the next access of that memory line. A reference with forward reuse distance $d \geq n$ will be removed from the cache before the next access.*

Proof. If the forward reuse distance is infinite, the data will not be used in the future, so there is no next access.

Consider the forward reuse distance of reference r_1 and assume that the next access to the data occurs at reference r_2, resulting in a reuse pair $\langle r_1, r_2 \rangle$. By definition, the forward reuse distance d of r_1 equals the backward reuse distance of r_2, i.e. d. Lemma 1 stipulates that the data will be found in the cache at reference r_2, if and only if $d < n$. □

Lemmas 1 and 2 indicate that the reuse distance can be used to precisely indicate the cache behavior of fully-associative caches. However, previous research[1] indicates that also for lower-associative, and even for direct mapped caches, the reuse distance can be used to obtain a good estimation of the cache behavior.

4 Cache Hint Selection

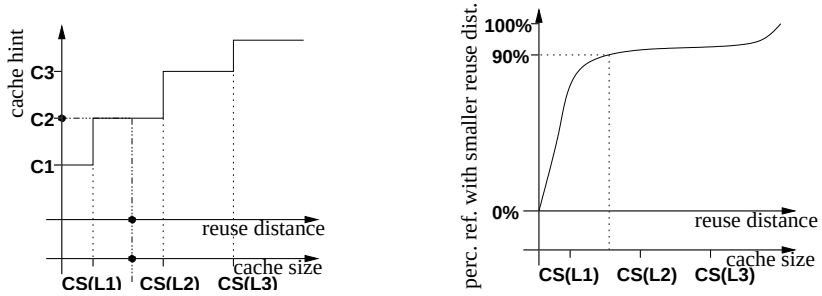
4.1 Reuse Distance-Based Selection

The cache hint selection is based on the forward and backward reuse distances of the accesses. Lemma 1 is used to select the most appropriate *source cache specifier* for a fully associative cache, i.e. the smallest and fastest cache level where data will be found upon reference. This is the smallest cache level with a size larger than the *backward reuse distance*. Similarly, lemma 2 yields the following *target cache specifier* selection: the specifier must indicate the smallest cache where the data will be found upon the next reference, i.e. the cache level with a size larger than the *forward reuse distance*. This mapping from reuse distance to cache hint is graphically shown in fig. 3(a).

For every memory access, the most appropriate cache hint can be determined. However, a single memory instruction can generate multiple memory accesses during program execution. Those accesses can demand different cache hints. It is not possible to specify different cache hints for them, since the cache hint is specified on the instruction. As a consequence, all accesses originating from the same instruction share the same cache hint. Because of this, it is not possible to assign the most appropriate cache hint to all accesses. In order to select a cache hint which is reasonable for most memory accesses generated by an instruction, we use a threshold value. In our experiments, the cache hint indicates the smallest cache level appropriate for at least 90% of the accesses, as depicted in fig. 3(b).

4.2 Cache Data Dependencies

The source cache specifier makes the compiler aware of the cache behavior. However, adding cache dependencies, in combination with source cache specifiers further refines the compilers view on the latency of memory instructions. Consider fig 4. Two loads access data from the same cache line in a short time period. The first load misses the cache. Since the first load brings the data into the fastest cache level, the second load hits the cache. However, the second load can only hit the cache if the first load had enough time to bring the data into



(a) Cache hint in function of the reuse distance of a single access.

(b) Cumulative reuse distance distribution (CDF) of an instruction The 90th percentile determines the cache hint.

Fig. 3. The selection of cache hints, based on the reuse distance. In (a), it is shown how the reuse distance of a single memory access maps to a cache level and an accompanying cache hint. For example, a reuse distance larger than the cache size of L1, but smaller than L2 results in cache hints C2. In (b), a cumulative reuse distance distribution for an instruction is shown and how a threshold value of 90% maps it to cache hint C2.

the cache. Therefore, the second load is cache dependent on the first load. If this dependence is not visible to the scheduler, it could schedule the second load with cache hit latency, before the first load has brought the data into the cache. This can lead to a schedule where the instructions dependent on the second load would be issued before their input data is available, leading to processor stall on an in-order EPIC machine.

One instruction can generate multiple accesses, with the different accesses coming from the same instruction dictating different cache dependencies. A threshold is used to decide if an instruction is cache dependent on another instruction. If a load instruction y accesses a memory line at a certain cache level, and that memory line is brought to that cache level by instruction x in at least 5% of the accesses, a cache dependence from instruction x to instruction y is inserted.

5 Experiments

The Itanium processor, the first implementation of the IA-64 ISA, was chosen to test the cache hint selection scheme described above. The Itanium processor provides cache hints as described in sect. 2.

5.1 Implementation

The above cache hint selection scheme was implemented in the Open64 compiler[8], which is based on SGI's Pro64 compiler. The reuse distance distribution for the

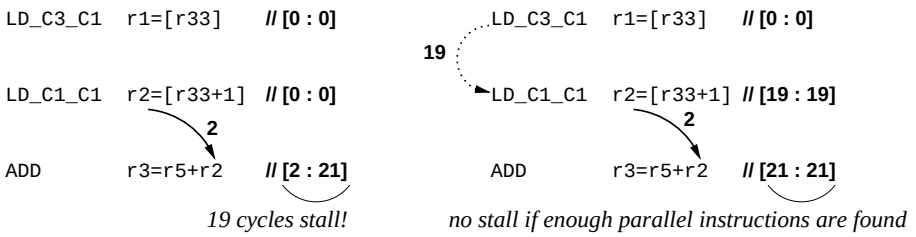


Fig. 4. An example of the effect of cache dependence edges in the instruction scheduler. The two load instructions access the same memory line. The first number between square brackets indicates the schedulers idea of the first cycle in which the instruction can be executed. The second number shows the real cycle in which the instruction can be executed. On the left, there is no cache dependence edge and a stall of up to 19 cycles can occur, while the instruction scheduler is not aware of it. On the right hand, the cache dependence is visible to the compiler, and the scheduler can try to move parallel instruction between the first and the second load instruction to hide the latency.

memory instructions, and the necessary information needed to create cache dependencies are obtained by instrumenting and profiling the program. The source and target cache hints are annotated to the memory instruction, based on the profile data. After instruction scheduling, the compiler produces the EPIC assembly code with target cache hints.

All compilations were performed at optimization level -O2, the highest level at which instrumentation and profiling is possible in the Open64 compiler. The existing framework doesn't allow to propagate the feedback information through some optimizations phases at level -O3.

5.2 Measurements

The programs were executed on a HP rx4610 multiprocessor, equipped with 733MHz Itanium processors. The data cache hierarchy consists of a 16KB L1, 96KB L2 and a 2MB L3 cache. The hardware performance counters of the processor were used to obtain detailed micro-architectural information, such as processor stall time because of memory latency and cache miss rates.

The programs were selected from the Olden and the Spec95fp benchmarks. The Olden benchmark contains programs which uses dynamic data structures, such as linked lists, trees and quadtrees. The Spec95fp programs are numerical programs with mostly regular array accesses. For the Spec95fp, the profiling was done using the train input sets, while the speedup measurements were done with the large input sets. For Olden, no separate input sets are available, and the training input was identical to the input for measuring the speedup. The results of the measurements can be found in table 1.

The table shows that the programs run 7% faster on average, with a maximum execution time reduction of 36%. In the worst case, a slight performance

Table 1. Table with results for programs from the Olden and the SPEC95FP benchmarks: mem. stall=percentage of time the processor stalls waiting for the memory; mem. stall reduction=the percentage of memory stall time reduction after optimization; source CH speedup=the speedup if only source cache specifiers are used; target CH speedup=speedup if only target cache specifiers are used; missrate reduction=reduction in miss rate for the three cache levels; overall speedup=speedup resulting from reuse distance-based cache hint selection.

			mem. stall	source CH	target CH	missrate reduction			overall
	program	mem. stall	reduction	speedup	speedup	L1	L2	L3	speedup
Olden	bh	26%	0%	0%	-1%	1%	-20%	-3%	**-1%**
	bisort	32%	0%	0%	0%	0%	6%	-5%	**0%**
	em3d	77%	25%	6%	20%	-28%	-3%	35%	**23%**
	health	80%	19%	2%	16%	0%	-1%	15%	**20%**
	mst	72%	1%	0%	0%	-10%	1%	2%	**1%**
	perimeter	53%	-1%	-1%	-1%	-11%	-56%	-6%	**-2%**
	power	15%	0%	0%	0%	-14%	2%	0%	**0%**
	treeadd	48%	0%	-2%	-1%	-2%	26%	17%	**0%**
	tsp	20%	0%	0%	0%	2%	7%	7%	**0%**
	Olden avg.	47%	5%	0%	4%	-6%	-6%	7%	**5%**
Spec95fp	swim	78%	0%	0%	1%	32%	0%	0%	**0%**
	tomcatv	69%	33%	7%	4%	-11%	-43%	6%	**9%**
	applu	49%	10%	4%	1%	-9%	-1%	-1%	**4%**
	wave5	43%	-9%	4%	15%	-26%	-7%	-5%	**5%**
	mgrid	45%	13%	36%	0%	13%	-24%	25%	**36%**
	Spec95fp avg.	57%	9%	10%	4%	0%	-15%	5%	**10%**
	overall avg.	51%	7%	4%	4%	-5%	-8%	6%	**7%**

degradation of 2% is observed. On average, the Olden benchmarks do not profit from the source cache specifiers. To take advantage of the source cache specifiers, the instruction scheduler must be able to find parallel instructions to fit in between a long latency load and its consuming instructions. In the pointer-based Olden benchmarks, the scheduler finds little parallel instructions, and cannot profit from its better view on the cache behavior.On the other hand, in the floating point programs, on average a 10% speedup is found because of the source cache hints. Here, the loop parallelism allows the compiler to find parallel instructions, mainly because it allows it to software pipeline the loops with long latency loads. In this way, the latency is overlapped with parallel instructions from different loop iterations. Some of the floating point programs didn't speedup a lot when employing source cache specifiers. The scheduler couldn't generate better code since the long latency of the loads demanded too many software pipeline stages to overlap it. Because of the large number of pipeline stages, not enough registers were available to actually create the software pipelining code.

The table also shows that the target cache specifiers improve both kind of programs by the same percentage. This improvement is caused by an average reduction in the L3 cache misses of 6%. The reduction is due to the improved

cache replacement decisions made by the hardware, based on the target cache specifiers.

6 Related Work

Much work has been done to eliminate cache misses by loop and data transformations. In our approach, the remaining cache misses after these transformations are further diminished in two orthogonal ways: target cache specifiers and source cache specifiers. In the literature, ideas similar to either the target cache specifier or the source cache specifier are proposed, but not both.

Work strongly related to target cache specifiers is found in [5], [11], [13] and [14]. In [13], it is shown that less than 5% of the load instructions cause over 99% of all cache misses. In order to improve the cache behavior, the authors propose not allocating the data in the cache when the instruction has a low hit ratio. This results in a large decrease of the memory bandwidth requirement, while the hit ratio drops only slightly. In [5], keep and kill instructions are proposed. The keep instruction locks data into the cache, while the kill instruction indicates it as the first candidate to be replaced. Jain et al. also proof under which conditions the keep and kill instructions improve the cache hit rate. In [14], it is proposed to extend each cache line with an EM(Evict Me)-bit. The bit is set by software, based on compiler analysis. If the bit is set, that cache line is the first candidate to be evicted from the cache. In [11], a cache with 3 modules is presented. The modules are optimized respectively for spatial, temporal and spatial-temporal locality. The compiler indicates in which module the data should be cached, based upon compiler analysis or a profiling step. These approaches all suggest interesting modifications to the cache hardware, which allow the compiler to improve the cache replacement policy. However, the proposed modifications are not available in present day architectures. The advantage of our approach is that it uses cache hints available in existing processors. The results show that the presented cache hint selection scheme is able to increase the performance on real hardware.

The source cache specifiers hide the latency of cache misses. Much research has been performed on software prefetching, which also hides cache miss latency. However, prefetching requires extra prefetch instructions to be inserted in the program. In our approach, the latency is hidden without inserting extra instructions. Latency hiding without prefetch instructions is also proposed in [3] and [9]. In [3] the cache behavior of numerical programs is examined using miss traffic analysis. The detected cache miss latencies are hidden by techniques such as loop unrolling and shifting. In comparison, our technique also applies to non-numerical programs and the latencies are compensated by scheduling low level instructions. The same authors also introduce cache dependency, and propose to shift data accesses with cache dependencies to previous iterations. In the present paper, cache dependencies are treated as ordinary data dependencies. In [9], load instructions are classified into normal, list and stride access. List and stride accesses are maximally hidden by the compiler because they cause most

cache misses. However the classification of memory accesses in two groups is very coarse. The reuse distance provides a more accurate way to measure the data locality, and as such permits the compiler to generate a more balanced schedule.

Finally, all the approaches mentioned above apply only to a single cache level. In contrast, reuse distance based cache hint selection can easily be applied to multiple cache levels.

7 Conclusion

Cache hints emerge in new processor architectures. This opens the perspective of new optimization schemes aimed at steering the cache behavior from the software level. In order to generate appropriate cache hints, the data locality of the program must be measured. In this paper, the reuse distance is proposed as an effective locality metric. Since it is independent of cache parameters such as cache size or associativity, the reuse distance can be used for optimizations which target multiple cache levels. The properties of this metric allow a straightforward generation of appropriate cache hints. The cache hint selection was implemented in an EPIC compiler for Itanium processors. The automatic selection of source and target cache specifiers resulted in an average speedup of 7% in a number of integer and numerical programs, with a maximum speedup of 36%.

References

[1] K. Beyls and E. H. D'Hollander. Reuse distance as a metric for cache behavior. In *Proceedings of PDCS'01*, 2001.

[2] S. Ghosh. *Cache Miss Equations: Compiler Analysis Framework for Tuning Memory Behaviour*. PhD thesis, Princeton University, November 1999.

[3] P. Grun, N. Dutt, and A. Nicolau. MIST: An algorithm for memory miss traffic management. In *ICCAD*, 2000.

[4] *IA-64 Application Developer's Architecture Guide*, May 1999.

[5] P. Jain, S. Devadas, D. Engels, and L. Rudolph. Software-assisted replacement mechanisms for embedded systems. In *CCAD'01*, 2001.

[6] G. Kane. *PA-RISC 2.0 architecture*. Prentice Hall, 1996.

[7] V. Kathail, M. S. Schlansker, and B. R. Rau. HPL_PD architecture specification: Version 1.1. Technical Report HPL-93-80(R.1), Hewlett-Packard, February 2000.

[8] Open64 compiler. http://sourceforge.net/projects/open64.

[9] T. Ozawa, Y. Kimura, and S. Nishizaki. Cache miss heuristics and preloading techniques for general-purpose programs. In *MICRO'95*.

[10] K. R.E. The alpha 21264 microprocessor. *IEEE Micro*, pages 24–36, mar 1999.

[11] J. Sanchez and A. Gonzalez. A locality sensitive multi-module cache with explicit management. In *Proceedings of the 1999 Conference on Supercomputing*.

[12] H. Sharangpani and K. Arora. Itanium processor microarchitecture. *IEEE Micro*, 20(5):24–43, Sept./Oct. 2000.

[13] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A modified approach to data cache management. In *MICRO'95*.

[14] Z. Wang, K. McKinley, and A. Rosenberg. Improving replacement decisions in set-associative caches. In *Proceedings of MASPLAS'01*, April 2001.